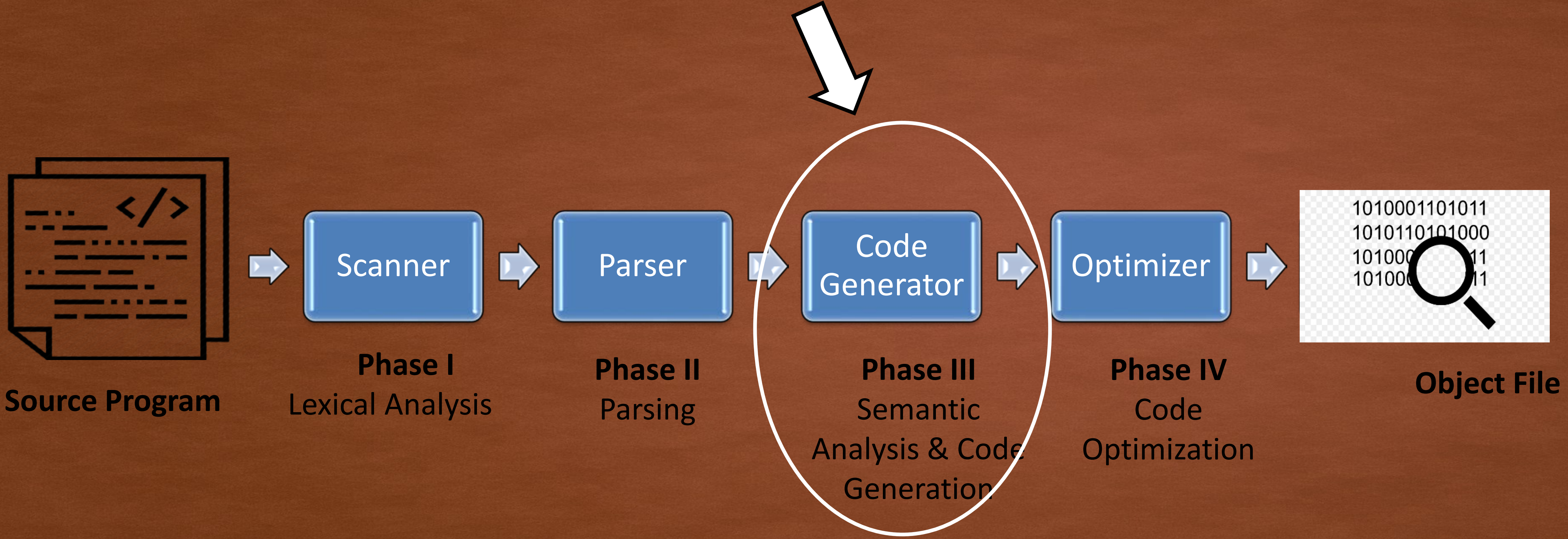
Phase III : Semantic Analysis and Code Generation



Phase III : Semantic Analysis

In this phase, the compiler examines the semantics of a programming language statement to determine whether it makes sense or not.

What does it mean for a program to be semantically valid?

Examples:

1. All variables, functions, classes, etc. are properly defined (Label Checking).
2. Expressions and variables are used in ways that respect the type system (Type Checking)
   - For example,
     - int a; string b;
     - a = b+5; $\longrightarrow$ Wrong!
3. Access control is respected (Flow control checks).
4. In languages where variables and parameters have types, such as C/C++/C# or Java, parameters & return types of a function must match.

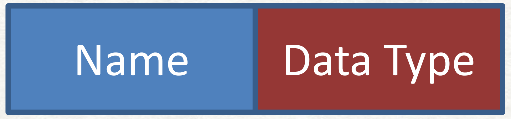
Phase III : Semantic Analysis

To check the semantic correctness of code, the compiler makes use of **semantic records** that are associated with each non terminal symbol in the grammar, such as `<expression>` and `<variable>`.

A semantic record is a sort of "declaration" record for each non-terminal symbol that the parser creates after parsing its declaration.

Semantic records

A *semantic record* is a data structure that records the following information for a non-terminal symbol :



A semantic record

It might also record other information (such as the memory address where the symbol’s value is stored).

Non-terminals with names

variables and function (method) names.

Non-terminals without names (or compiler-assigned pseudo-names)

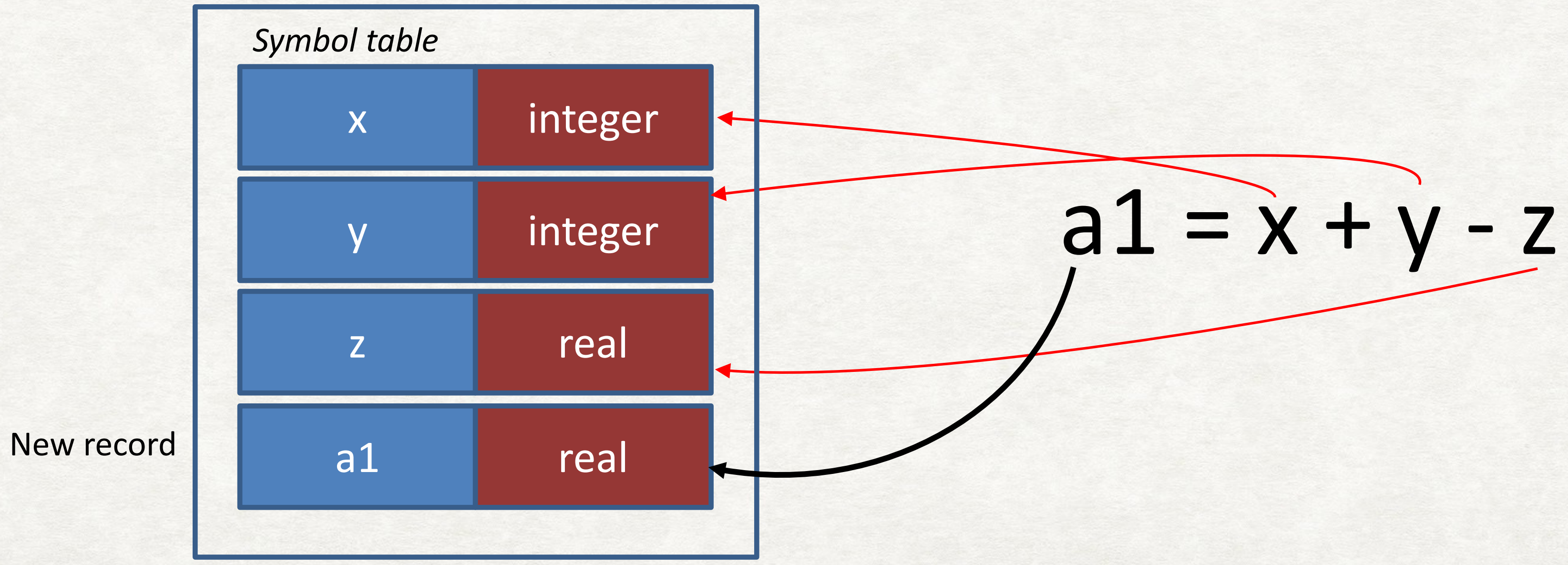
sums, products, assignment statements, etc.

How are semantic records assigned?

Semantic records are stored in a ***symbol table***.

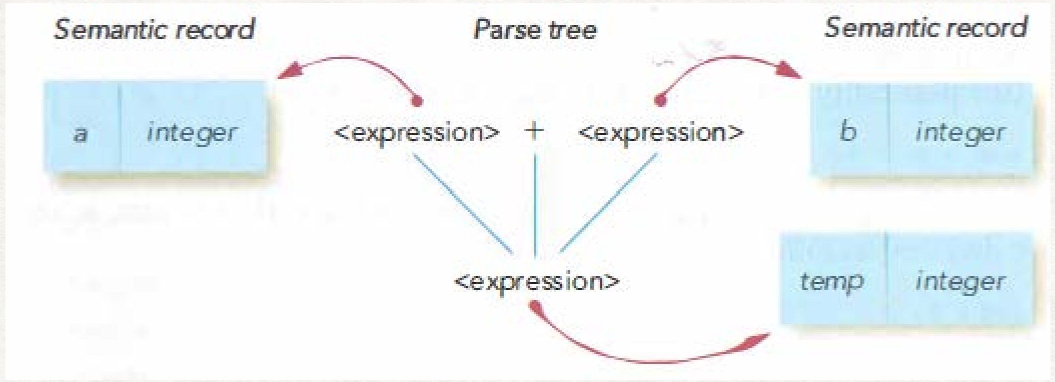
Traverse the parsing tree and assign semantic records as follows:

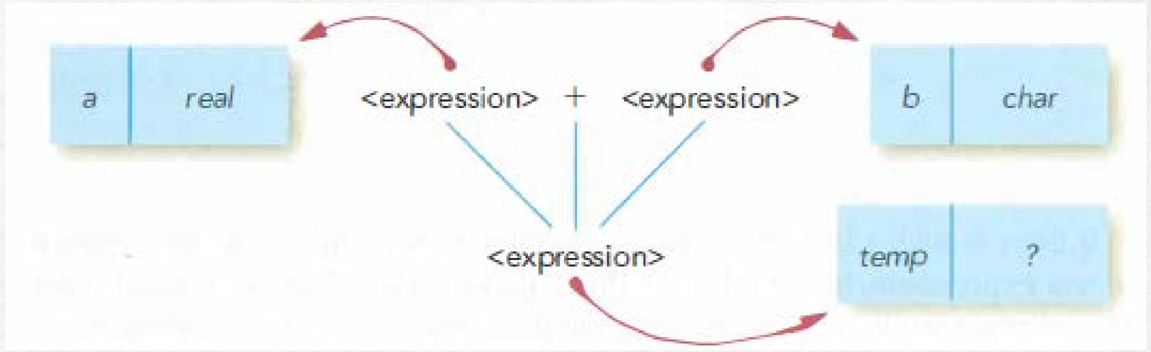
Every time the parser generates a new non-terminal, it links it to a semantic record.
If there is an existing semantic record for a non-terminal; link the non-terminal to the existing record.
If the non-terminal cannot be associated with an existing semantic record, generate a new semantic record and link the non-terminal to it.



How are semantic records used?

We use semantic records to ensure that a parsing tree ***makes sense.***

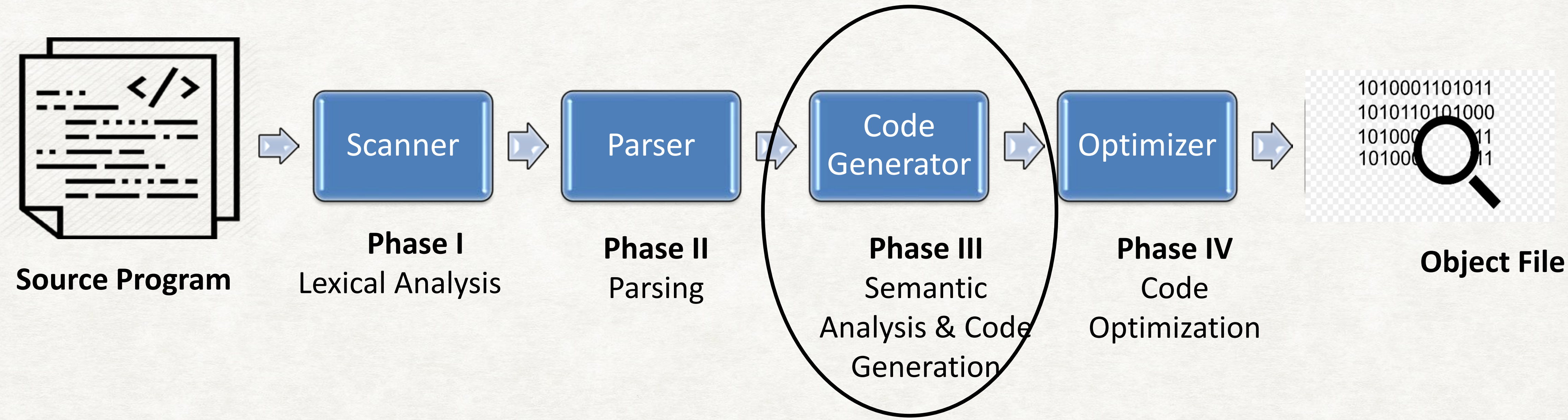




This parse tree says that we are adding two <expressions> that are of type *integer.*
Addition is well defined for integers, therefore this parse tree is *"semantically correct".*
The compiler can generate code to carry out this addition.

This parse tree says that we are adding two <expressions> one of which is a *real* and the other is a *char.*
Addition is not defined for a real number and a char, therefore this parse tree is *"semantically incorrect".*
The compiler cannot generate code to carry out this addition.

Phase III : Semantic Analysis and Code Generation



- First part : Semantic Analysis
  - Involves a pass over the parse tree to check whether all branches of the tree are semantically valid.
  - If so, then the compiler generates the machine code.
  - If not, then a “compile-time error” is raised and no code is generated.

- Second Part : Code Generation
  - Involves a second pass over the parse tree to produce the machine code.
  - Each production in the parse tree represents an action and the compiler must determine how this can be done in machine code.

How are semantic records used?

We also use semantic records in ***code generation***.

For example:

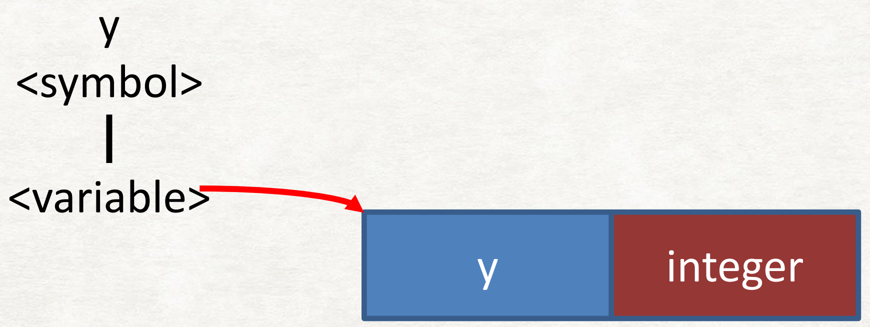
- When adding two variables, the name and type of the variable (e.g., integer or real) determines
    1. the address in memory that the machine language must copy the value of each variable from.
    2. how many bytes must be copied.
    3. the exact algorithm that will be coded (e.g., floating point addition vs. integer addition).

- For functions (methods), the semantic record determines the address in memory at which the machine must continue program execution.

Code generation normally starts at the leaves of the parsing tree and moves towards the root, one production at a time.

However, not all productions generate machine code as we will see.

Example : production recognising a symbol as a variable



Assembly code generated if y has no existing semantic record:

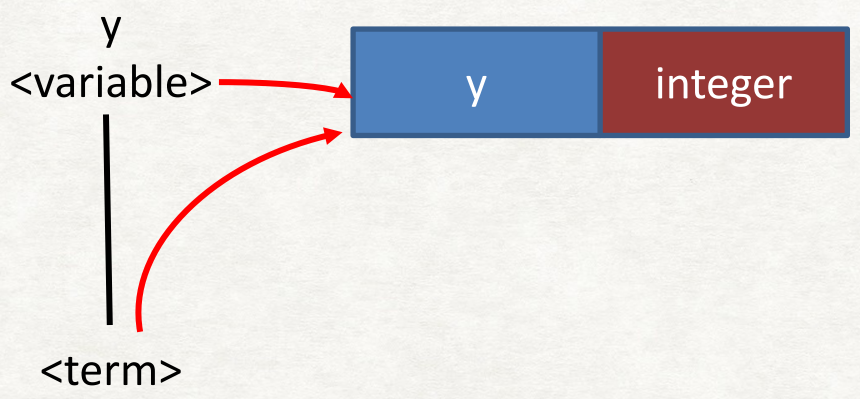
```
y:  .DATA 0
```

This may later be overwritten with any value we may assign to y.

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: production recognising a <variable> as a <term>.

Possibility 1: link new non-terminal to existing semantic record

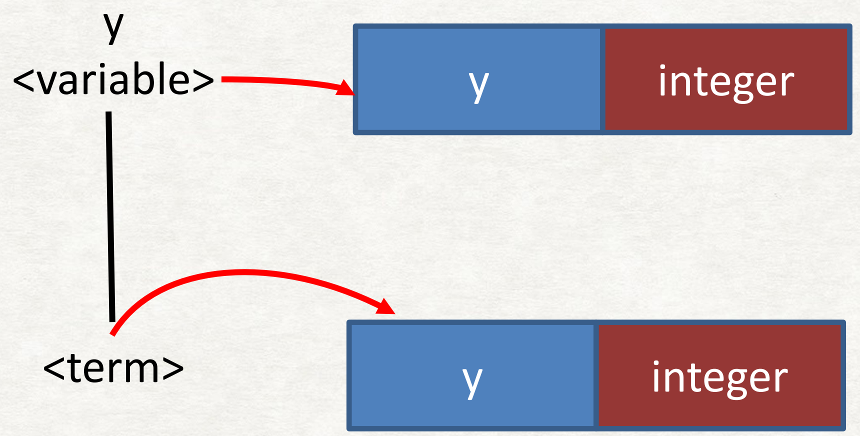


No machine code is generated for this production.

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Production recognising a <variable> as a <term>.

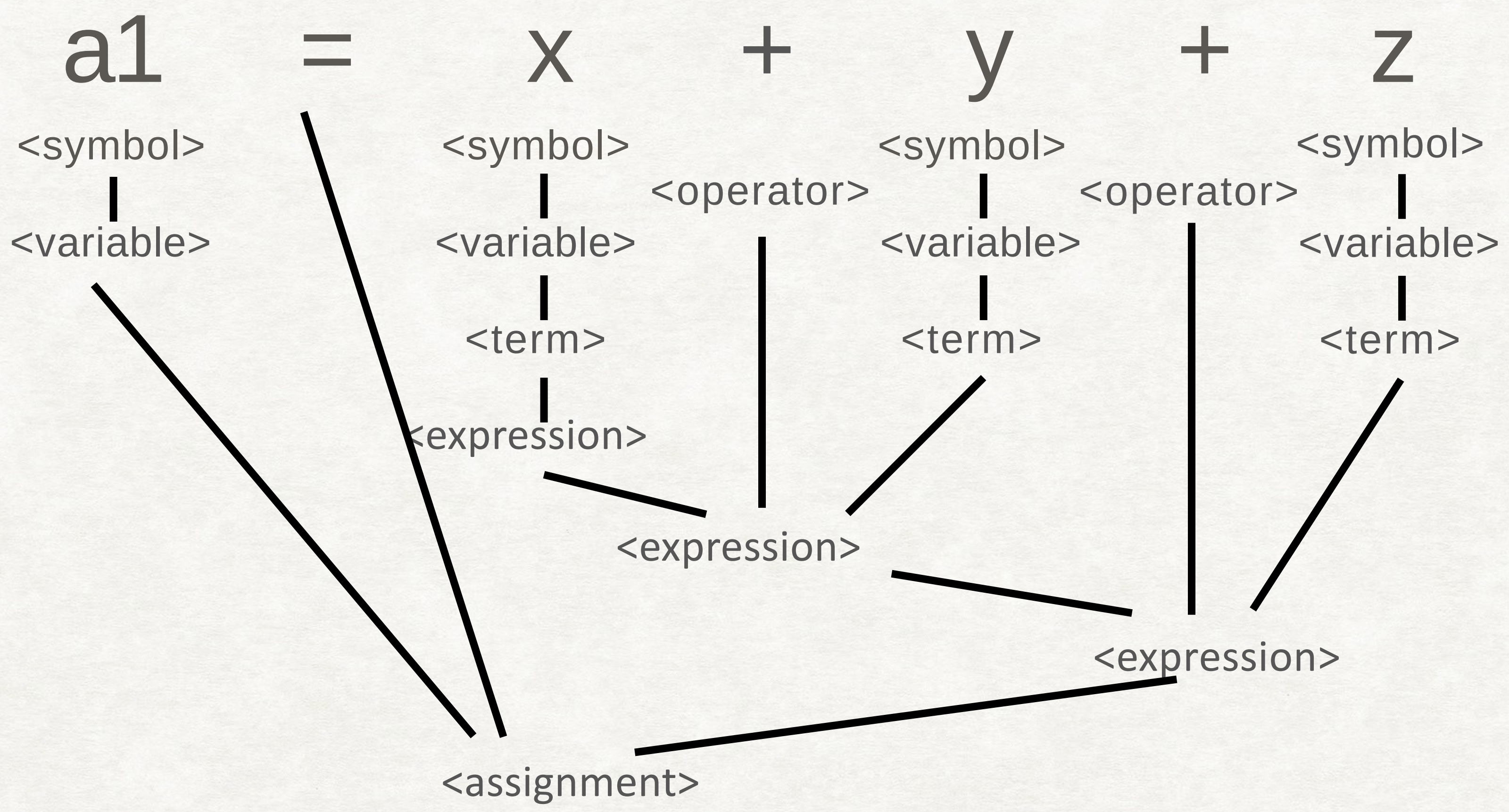
Possibility 2: create a new, identical semantic record for the new non-terminal



No machine code is generated for this production.

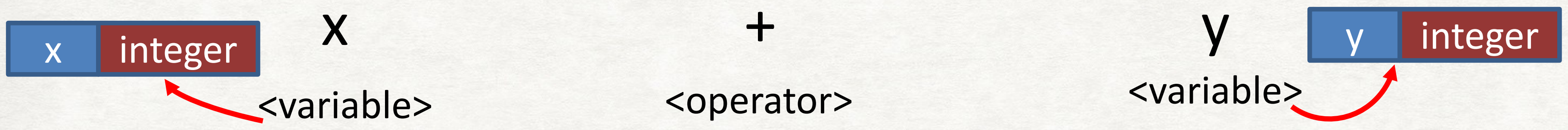
<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Let’s look at an example parse tree…



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Production combining two variables and an addition operator.

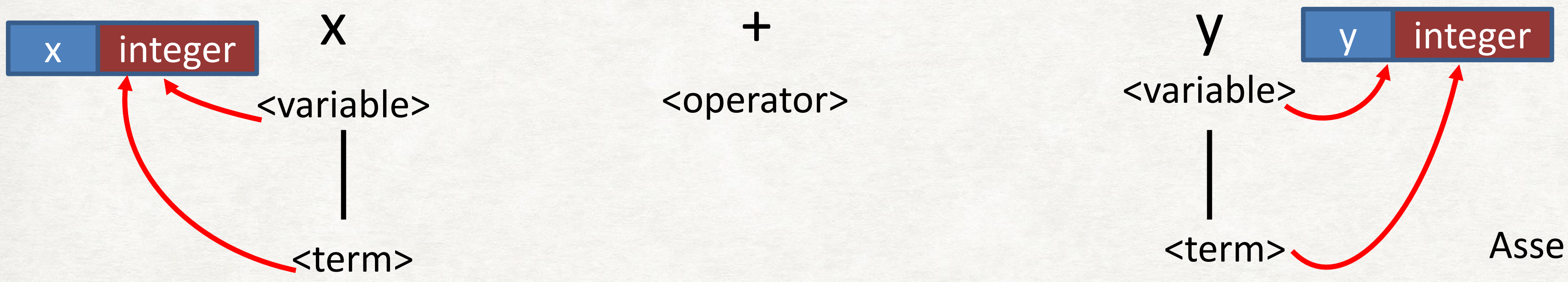


Assembly code generated:

```
x: .DATA 0
y: .DATA 0
```

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Production combining two variables and an addition operator.

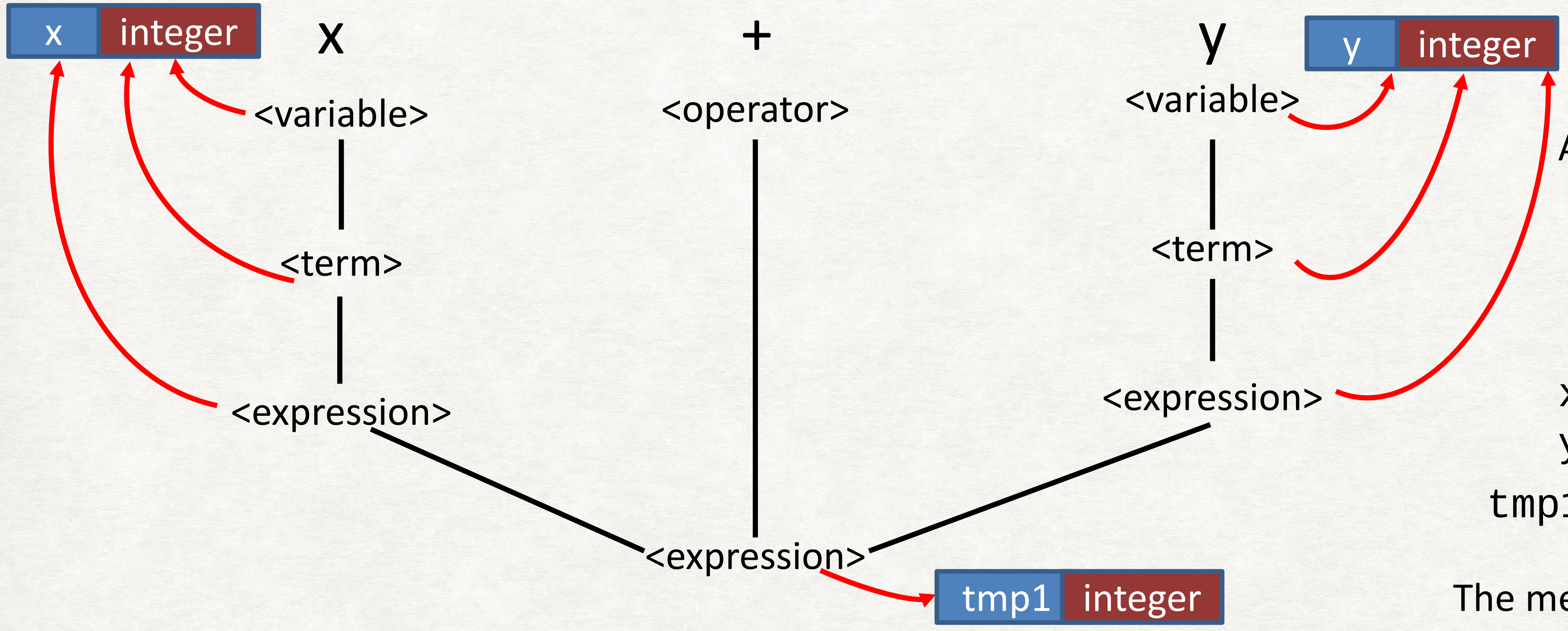


Assembly code generated:

```
x:  .DATA 0
y:  .DATA 0
```

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Production combining two variables and an addition operator.



Assembly code generated:

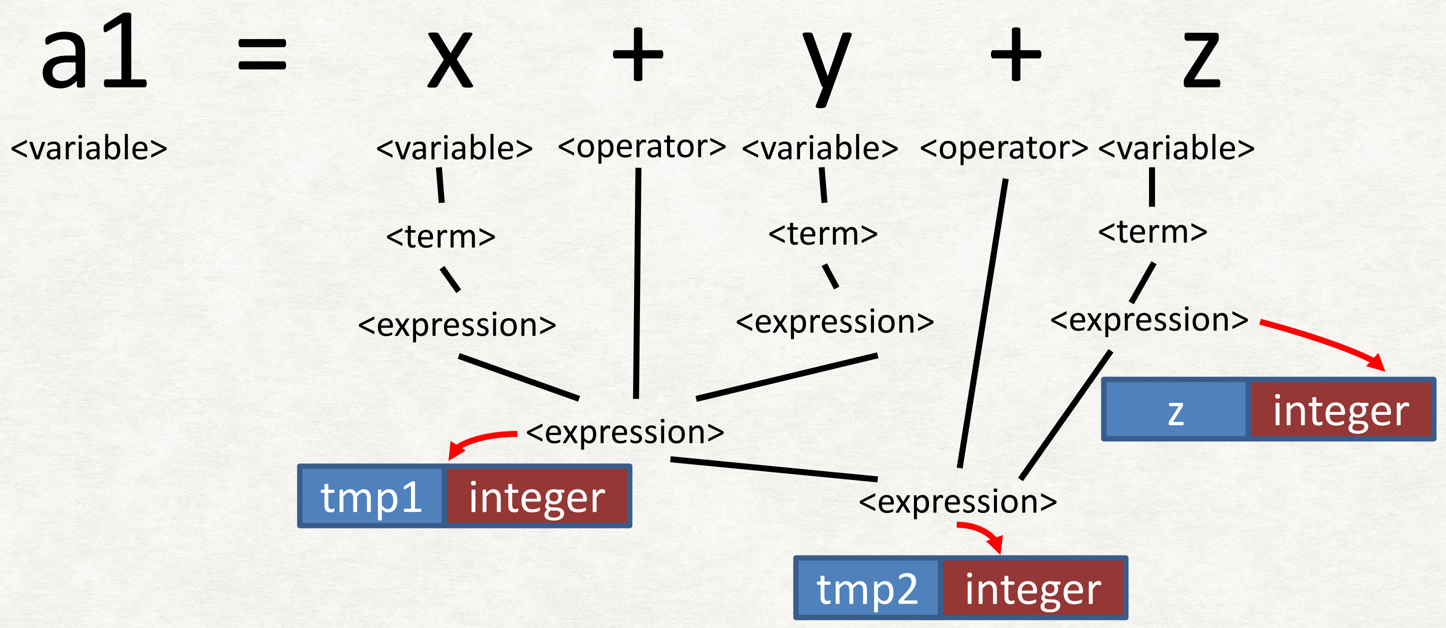
```
LOAD     x
ADD      y
STORE    tmp1

  x:  .DATA 0
  y:  .DATA 0
tmp1: .DATA 0
```

The memory cell named `tmp1` is automatically selected by the compiler to hold the result of the expression.

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Full assignment statement



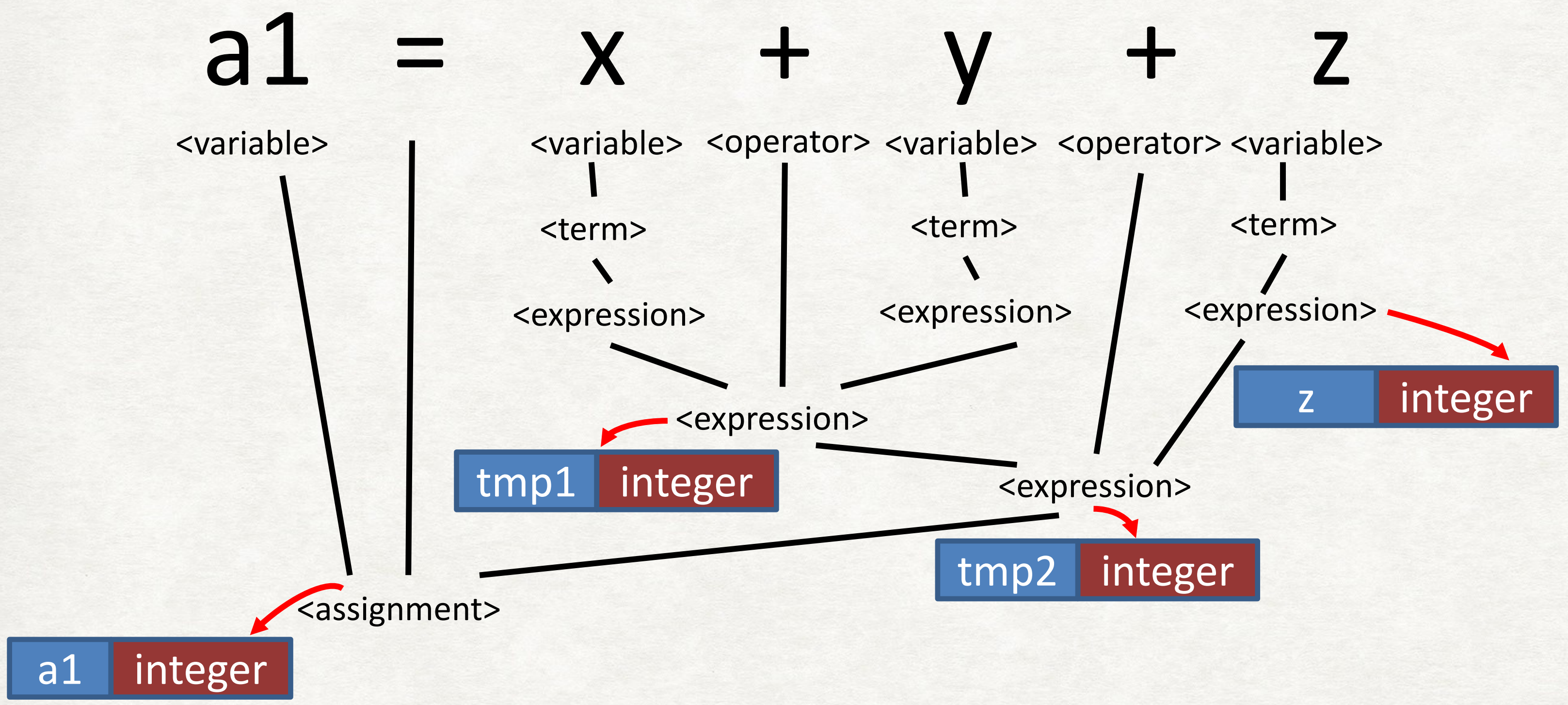
Assembly code generated:

```
LOAD    x
ADD     y
STORE   tmp1
LOAD    tmp1
ADD     z
STORE   tmp2

   x: .DATA 0
   y: .DATA 0
   z: .DATA 0
tmp1: .DATA 0
tmp2: .DATA 0
```

<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Example: Full assignment statement



<variable> ::= <symbol>
<operator> ::= +|-|*|/
<term> ::= <number>|<variable>
<expression> ::= <term>|<expression><operator><expression>
<assignment> ::= <variable>=<expression>

Assembly code generated:

```
    LOAD     x
    ADD      y
    STORE    tmp1
    LOAD     tmp1
    ADD      z
    STORE    tmp2
    LOAD     tmp2
    STORE    a1
  x:  .DATA 0
  y:  .DATA 0
  z:  .DATA 0
 a1:  .DATA 0
tmp1: .DATA 0
tmp2: .DATA 0
```

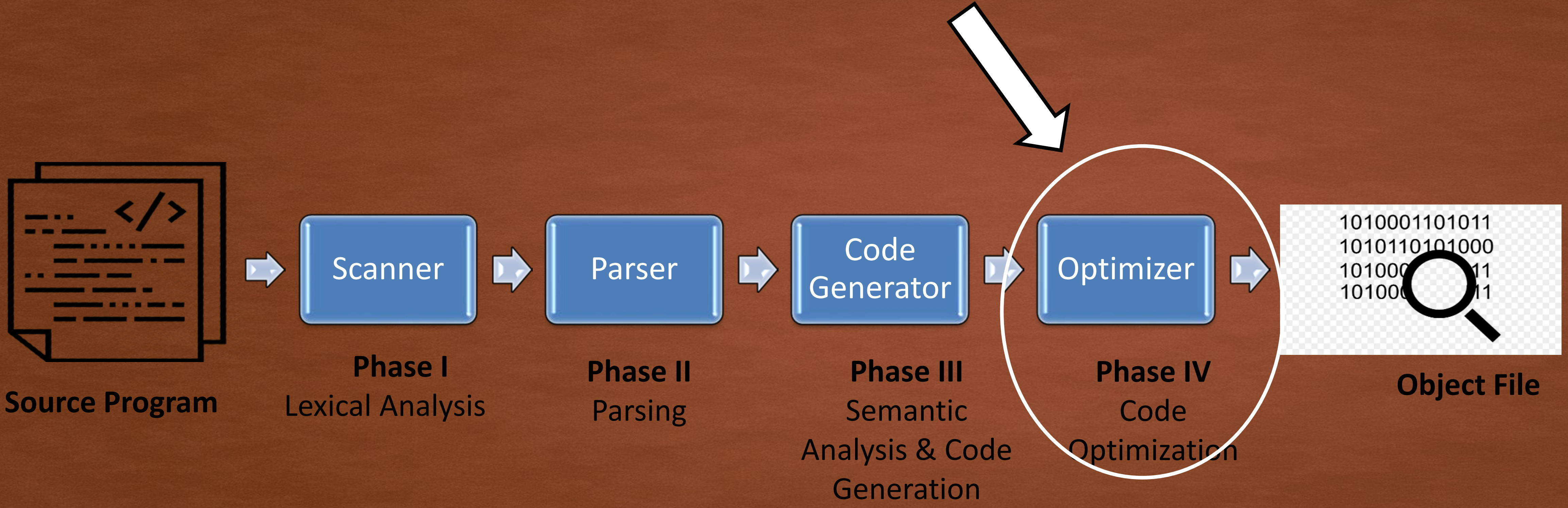
Summary

Semantic records keep track of variable names / functions / methods / expressions and their types, as well as other information as needed.

Semantic records ensure that syntactically correct code is also meaningful and can be translated into correct machine code.

Code generation starts from the leaves of the parsing tree, successively progressing down to the goal symbol.

Phase IV : Code Optimization



Phase IV: Code optimization

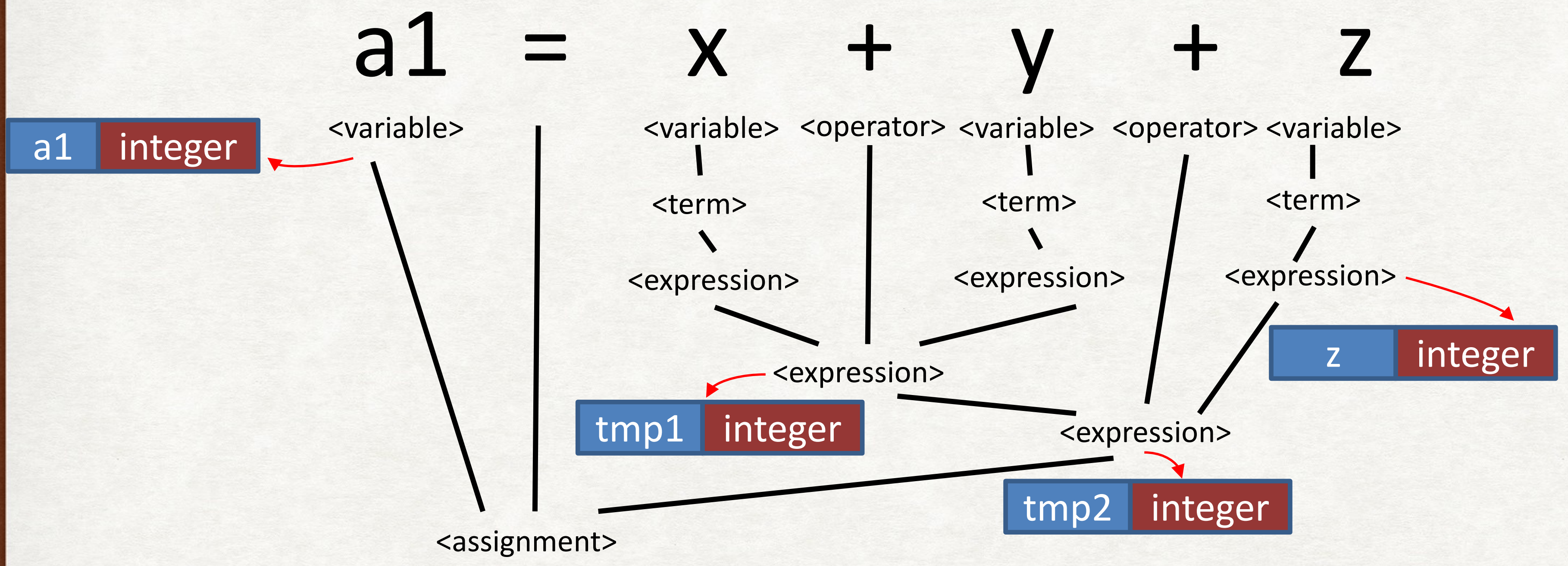
Compiler tries to improve the time or space efficiency of the generated machine code.

Early days: “Programmers are cheap, hardware is expensive”

- Humans could write more optimal code than a compiler

These days: “Hardware is cheap, people are expensive”

Example: Full assignment statement



Assembly code generated:

```
LOAD     x
ADD      y
STORE    tmp1
LOAD     tmp1
ADD      z
STORE    tmp2
LOAD     tmp2
STORE    a1

......
x:      .DATA 0
y:      .DATA 0
z:      .DATA 0
a1:     .DATA 0
tmp1:   .DATA 0
tmp2:   .DATA 0
```

What’s wrong with the code we just generated?

```
  LOAD      x
  ADD       y
  STORE     tmp1
  LOAD      tmp1
  ADD       z
  STORE     tmp2
  LOAD      tmp2
  STORE     a1

......
x:      .DATA 0
y:      .DATA 0
z:      .DATA 0
a1:     .DATA 0
tmp1:   .DATA 0
tmp2:   .DATA 0
```

- The code works.
- But: We use memory cells to store the value of expressions when this value is already in the register at the time we need it.
- We don’t need the additional STORE and LOAD operations.
- They only slow down code execution.

Code optimisation

- Remove the surplus STORE and LOAD operations.
- Fewer instructions -> faster program
- Fewer memory cells used

```
        LOAD      x
        ADD       y
        STORE     tmp1
        LOAD      tmp1
        ADD       z
        STORE     tmp2
        LOAD      tmp2
        STORE     a1

.....
x:     .DATA 0
y:     .DATA 0
Z:     .DATA 0
a1:    .DATA 0
tmp1:  .DATA 0
tmp2:  .DATA 0
```

```
LOAD      x
ADD       y
ADD       z
STORE     a1

.....
x:     .DATA 0
y:     .DATA 0
Z:     .DATA 0
a1:    .DATA 0
```

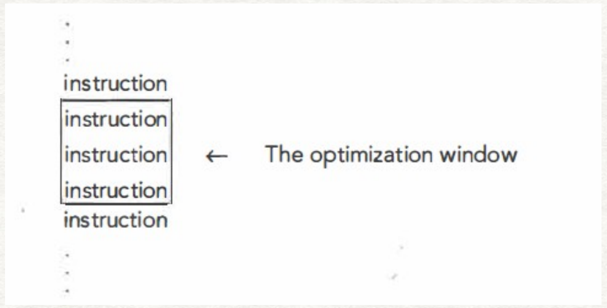
Code optimisation

The example on the previous slide is only one way in which a compiler can optimise code

There are many more ways which can broadly be classified into :

Local Optimization

Examines small chunks of assembly code (one to five instructions).



Global Optimization

Looks at large segments of the program

Examines blocks like while loops, if statements, and procedures

Much more difficult, much bigger effect

Local Optimization Examples

1. **Eliminating unnecessary operations:** removes operations that are unnecessary, like a LOAD of a value already in memory

2. **Constant evaluation:** computes arithmetic expressions at compile time if possible.
   e.g., turn x = 15/3*y into x=5*y.
   Works even better with floating point numbers.

3. **Strength reduction:** uses faster arithmetic alternatives
   e.g., 2 * x is equivalent to x + x,
   or, x=x+3 is equivalent to INCREMENT X three times

Global Optimization Example

A much harder problem is trying to optimize over larger sections of the program.

E.g. Removing Operations from inside loops

```
// Imagine a loop which looks like this
answer = 0
while i < 1000000
      // Multiply each element of an array arr with 4.235 and sum them together:
      answer = answer + 4.235 * arr[i]
      i = i + 1
end while
```

Multiplication **inside** loop
= 1000000 multiplication operations!

```
// This can be done more efficiently as follows:
answer = 0
while i < 1000000
      // sum all the elements of the array first:
      answer = answer + arr[i]
      i = i + 1
end while
answer = 4.235 * answer
```

Multiplication **outside** loop
= 1 multiplication operation!

Global Optimization Example

Removing operations from loops

Consider the following pseudocode:

```
// Computing distance from Auckland to Wellington and back
stops = {Hamilton, Rotorua, Taupo, PalmerstonNorth}
start = Auckland
distance = 0
for (stop in stops) {
      // Add 2 * hop distance because we need to travel each leg
      // in both directions:
      distance = distance + 2 * DistanceBetween(start, stop)
      start = stop
}
Output distance
```

Multiplication **inside** loop
= 4 multiplication operations!

Global Optimization Example

Removing operations from loops

Consider the following pseudocode:

```
// Computing distance from Auckland to Wellington and back
stops = {Hamilton, Rotorua, Taupo, PalmerstonNorth}
start = Auckland
distance = 0
for (stop in stops) {
      // Add 2 * hop distance because we need to travel each leg
      // in both directions:
      distance = distance + DistanceBetween(start, stop)
      start = stop
}
Output 2 * distance
```

Multiplication **outside** loop
= only 1 multiplication operation!

Summary

Code generated by simply translating each production in the parse tree is not necessarily efficient.

Good compilers will optimise.

Optimization is a trade-off between:

- Effort put into enabling a compiler to recognise and fix inefficiencies.
- Compile time (compilers that have to look for inefficiencies are slower).
- Speed of the resulting program.

Side note 1

Real compilers combine some of the four phases of compilation, so they run almost concurrently:

Parsing often starts the moment the lexical analysis identifies the first token, and continues each time the lexer identifies a new token.

Semantic analysis and code generation for a non-terminal happen immediately after successful productions.

The compiler performs optimisation whenever code is generated – not just at the end.

Side note 2

Code optimization **cannot** make an inefficient algorithm efficient.

The efficiency of an algorithm is an inherent characteristic of its structure.
It is not something programmed in by a programmer,
Or, optimized by a compiler.

A sequential search program compiled by an optimizing compiler will still run much slower than a binary search program written by a novice programmer and compiled using a non-optimizing compiler.